SQL

STEP
BY
STEP

BASIC ADVANCE

FUNCTIONS

Functions In SQL

Functions in SQL are the database objects that contains a set of SQL statements to perform a specific task. A function accepts input parameters, perform actions, and then return the result.

Types of Function:

1. **System Defined Function** : these are built-in functions

   - **Scalar functions** → operate on single values (numeric (abs), string, date).
   - **Aggregate functions** → operate on sets of rows.
   - Example: rand(), round(), upper(), lower(), count(), sum(), avg(), max(), etc

2. **User-Defined Function** : Once you define a function, you can call it in the same way as the built-in functions

Most Used Aggregate Functions

Aggregate function performs a calculation on multiple values (rows) and returns a single value.

And Aggregate functiona are often used with GROUP BY & SELECT statement

- **COUNT()** returns number of values
- **SUM()** returns sum of all values
- **AVG()** returns average value
- **MAX()** returns maximum value
- **MIN()** returns minimum value
- **ROUND()** Rounds a number to a specified number of decimal places

Most Used String Functions

String functions are used to perform an operation on input string and return an output string

- **UPPER()** converts the value of a field to uppercase
- **LOWER()** converts the value of a field to lowercase
- **LENGTH()** returns the length of the value in a text field
- **SUBSTRING()** extracts a substring from a string
- **NOW()** returns the current system date and time
- **FORMAT()** used to set the format of a field
- **CONCAT()** adds two or more strings together
- **REPLACE()** Replaces all occurrences of a substring within a string, with a new substring
- **TRIM()** removes leading and trailing spaces (or other specified characters) from a string

Payments_fun table

```
CREATE TABLE Payments_fun (payment_id INT PRIMARY KEY,
                          customer_name VARCHAR(50),
                           amount DECIMAL(10,2),
                          mode VARCHAR(30),
                           payment_date DATE);

INSERT INTO Payments_fun (payment_id, customer_name,
amount, mode, payment_date) VALUES(1, 'Arjun', 60, 'Cash',
'2020-09-24'),
          (2, 'Meera', 30, 'Credit Card', '2020-04-27'),
          (3, 'Ravi', 90, 'Credit Card', '2020-07-07'),
          (4, 'Sneha', 50, 'Debit Card', '2020-02-12'),
          (5, 'Kiran', 40, 'Mobile Payment', '2020-11-20'),
          (6, 'Priya', 40, 'Debit Card', '2021-06-28'),
          (7, 'Rahul', 10, 'Cash', '2021-08-25'),
          (8, 'Anita', 30, 'Mobile Payment', '2021-06-17'),
          (9, 'Vikram', 80, 'Cash', '2021-08-25'),
          (10, 'Divya', 50, 'Mobile Payment', '2021-11-03'),
          (11, 'Suresh', 70, 'Cash', '2022-11-01'),
          (12, 'Neha', 60, 'Netbanking', '2022-09-11'),
          (13, 'Ajay', 30, 'Netbanking', '2022-12-10'),
          (14, 'Varun', 50, 'Credit Card', '2022-05-14'),
          (15, 'Pooja', 30, 'Credit Card', '2022-09-25');
```

Solve these Using Payments_fun table

1. Find the absolute difference between the highest and lowest payment amounts.

2. Round the average payment amount to 2 decimal places

3. Show all customer names in uppercase.

4. Extract the first 3 letters of each payment mode.

5. Find all payments made in 2021.

6. Calculate the number of days between the earliest and latest payment

7. Count how many payments were made by Credit Card.

8. Find the total amount collected per payment mode

Functions Cheat Sheet

TEXT FUNCTIONS

CONCATENATION

Use the || operator to concatenate two strings:

```
SELECT 'Hi ' || 'there!';
-- result: Hi there!
```

Remember that you can concatenate only character strings using ||. Use this trick for numbers:

```
SELECT '' || 4 || 2;
-- result: 42
```

Some databases implement non-standard solutions for concatenating strings like CONCAT() or CONCAT_WS(). Check the documentation for your specific database.

LIKE OPERATOR – PATTERN MATCHING

Use the _ character to replace any single character. Use the % character to replace any number of characters (including 0 characters).

Fetch all names that start with any letter followed by 'atherine':

```
SELECT name
FROM names
WHERE name LIKE '_atherine';
```

Fetch all names that end with 'a':

```
SELECT name
FROM names
WHERE name LIKE '%a';
```

USEFUL FUNCTIONS

Get the count of characters in a string:

```
SELECT LENGTH('LearnSQL.com');
-- result: 12
```

Convert all letters to lowercase:

```
SELECT LOWER('LEARNSQL.COM');
-- result: learnsql.com
```

Convert all letters to uppercase:

```
SELECT UPPER('LearnSQL.com');
-- result: LEARNSQL.COM
```

Convert all letters to lowercase and all first letters to uppercase (not implemented in MySQL and SQL Server):

```
SELECT INITCAP('edgar frank ted cODD');
-- result: Edgar Frank Ted Codd
```

Get just a part of a string:

```
SELECT SUBSTRING('LearnSQL.com', 9);
-- result: .com
SELECT SUBSTRING('LearnSQL.com', 0, 6);
-- result: Learn
```

Replace part of a string:

```
SELECT REPLACE('LearnSQL.com', 'SQL',
'Python');
-- result: LearnPython.com
```

NUMERIC FUNCTIONS

BASIC OPERATIONS

Use +, -, *, / to do some basic math. To get the number of seconds in a week:

```
SELECT 60 * 60 * 24 * 7; -- result: 604800
```

CASTING

From time to time, you need to change the type of a number. The CAST() function is there to help you out. It lets you change the type of value to almost anything (integer, numeric, double precision, varchar, and many more).

Get the number as an integer (without rounding):

```
SELECT CAST(1234.567 AS integer);
-- result: 1234
```

Change a column type to double precision

```
SELECT CAST(column AS double precision);
```

USEFUL FUNCTIONS

Get the remainder of a division:

```
SELECT MOD(13, 2);
-- result: 1
```

Round a number to its nearest integer:

```
SELECT ROUND(1234.56789);
-- result: 1235
```

Round a number to three decimal places:

```
SELECT ROUND(1234.56789, 3);
-- result: 1234.568
```

PostgreSQL requires the first argument to be of the type numeric – cast the number when needed.

To round the number **up**:

```
SELECT CEIL(13.1); -- result: 14
SELECT CEIL(-13.9); -- result: -13
```

The CEIL(x) function returns the **smallest** integer **not less** than x. In SQL Server, the function is called CEILING().

To round the number **down**:

```
SELECT FLOOR(13.8); -- result: 13
SELECT FLOOR(-13.2); -- result: -14
```

The FLOOR(x) function returns the **greatest** integer **not greater** than x.

To round towards 0 irrespective of the sign of a number:

```
SELECT TRUNC(13.5); -- result: 13
SELECT TRUNC(-13.5); -- result: -13
```

TRUNC(x) works the same way as CAST(x AS integer). In MySQL, the function is called TRUNCATE().

To get the absolute value of a number:

```
SELECT ABS(-12); -- result: 12
```

To get the square root of a number:

```
SELECT SQRT(9); -- result: 3
```

NULLs

To retrieve all rows with a missing value in the price column:

```
WHERE price IS NULL
```

To retrieve all rows with the weight column populated:

```
WHERE weight IS NOT NULL
```

Why shouldn't you use price = NULL or weight != NULL? Because databases don't know if those expressions are true or false – they are evaluated as NULLs.

Moreover, if you use a function or concatenation on a column that is NULL in some rows, then it will get propagated. Take a look:

domain	LENGTH(domain)
LearnSQL.com	12
LearnPython.com	15
NULL	*NULL*
vertabelo.com	13

USEFUL FUNCTIONS

COALESCE(x, y, ...)

To replace NULL in a query with something meaningful:

```
SELECT
  domain,
  COALESCE(domain, 'domain missing')
FROM contacts;
```

domain	coalesce
LearnSQL.com	LearnSQL.com
NULL	domain missing

The COALESCE() function takes any number of arguments and returns the value of the first argument that isn't NULL.

NULLIF(x, y)

To save yourself from *division by 0* errors:

```
SELECT
  last_month,
  this_month,
  this_month * 100.0
    / NULLIF(last_month, 0)
    AS better_by_percent
FROM video_views;
```

last_month	this_month	better_by_percent
723786	1085679	150.0
0	178123	*NULL*

The NULLIF(x, y) function will return NULL if x is the same as y, else it will return the x value.

CASE WHEN

The basic version of CASE WHEN checks if the values are equal (e.g., if fee is equal to 50, then 'normal' is returned). If there isn't a matching value in the CASE WHEN, then the ELSE value will be returned (e.g., if fee is equal to 49, then 'not available' will show up.

```
SELECT
  CASE fee
    WHEN 50 THEN 'normal'
    WHEN 10 THEN 'reduced'
    WHEN  0 THEN 'free'
    ELSE 'not available'
  END AS tariff
FROM ticket_types;
```

The most popular type is the **searched CASE WHEN** – it lets you pass conditions (as you'd write them in the WHERE clause), evaluates them in order, then returns the value for the first condition met.

```
SELECT
  CASE
    WHEN score >= 90 THEN 'A'
    WHEN score >  60 THEN 'B'
    ELSE 'F'
  END AS grade
FROM test_results;
```

Here, all students who scored at least 90 will get an A, those with the score above 60 (and below 90) will get a B, and the rest will receive an F.

TROUBLESHOOTING

Integer division

When you don't see the decimal places you expect, it means that you are dividing between two integers. Cast one to decimal:

```
CAST(123 AS decimal) / 2
```

Division by 0

To avoid this error, make sure that the denominator is not equal to 0. You can use the NULLIF() function to replace 0 with a NULL, which will result in a NULL for the whole expression:

```
count / NULLIF(count_all, 0)
```

Inexact calculations

If you do calculations using real (floating point) numbers, you'll end up with some inaccuracies. This is because this type is meant for scientific calculations such as calculating the velocity. Whenever you need accuracy (such as dealing with monetary values), use the decimal / numeric type (or money if available).

Errors when rounding with a specified precision

Most databases won't complain, but do check the documentation if they do. For example, if you want to specify the rounding precision in PostgreSQL, the value must be of the numeric type.

GROUP BY & HAVING CLAUSE

GROUP BY Statement

The GROUP BY statement group rows that have the same values into summary rows.

It is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns

- **Syntax**

```
SELECT column_name(s)
FROM table_name
GROUP BY column_name(s);
```

- **Example**

```
SELECT mode, SUM(amount) AS total
FROM payment_fun
GROUP BY mode
```

HAVING Clause

The **HAVING** clause is used to apply a filter on the result of **GROUP BY** based on the specified condition.
The **WHERE** clause places conditions on the selected columns, whereas the **HAVING** clause places conditions on groups created by the **GROUP BY** clause

Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition(s)
GROUP BY column_name(s)
HAVING condition(s)
```

- **Example**

```
SELECT mode, COUNT(amount) AS total
FROM payment_fun
GROUP BY mode
HAVING COUNT(amount) >= 3
ORDER BY total DESC
```

Quick Assignment: 01

Order of execution in SQL:

SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT

?

TIMESTAMPS & EXTRACT

TIMESTAMP

The **TIMESTAMP** data type is used for values that contain both date and time parts

- **TIME** contains only time, format HH:MI:SS
- **DATE** contains on date, format YYYY-MM-DD
- **YEAR** contains on year, format YYYY or YY
- **TIMESTAMP** contains date and time, format YYYY-MM-DD HH:MI:SS
- **TIMESTAMPTZ** contains date, time and time zone

TIMESTAMP functions/operators

Below are the TIMESTAMP functions and operators in SQL:

- SHOW TIMEZONE
- SELECT NOW()
- SELECT TIMEOFDAY()
- SELECT CURRENT_TIME
- SELECT CURRENT_DATE

EXTRACT Function

The **EXTRACT()** function extracts a part from a given date value.

Syntax: SELECT **EXTRACT**(MONTH FROM date_field) FROM Table

- **YEAR**
- **QUARTER**
- **MONTH**
- **WEEK**
- **DAY**
- **HOUR**
- **MINUTE**
- **DOW** – day of week
- **DOY** – day of year